Topics:

- **Map** implemented as a Binary Search Tree (BST)
  - Starting with a dream: binary search in a linked list?
  - How our dream provided the inspiration for the BST
  - BST insert
  - Big-O analysis of BST
  - BST balance issues

- **Traversals**
  - Pre-order
  - In-order
  - Post-order
  - Breadth-first

- Applications of Traversals
BST Balance Strategies

We need to balance the tree (keep $O(\log N)$ instead of $O(N)$), how can we do that if the tree structure is decided by key insert order?
AVL rotations: BST-order-preserving movement of nodes

- Here is a Binary Search Tree whose keys I’m not going to show you
  - (but the nodes have colors/textures so you can tell them apart)
- Let’s pause and think about what we know must be true
AVL rotations

- Here is a Binary Search Tree whose keys I’m not going to show you
  - (but the nodes have colors/textures so you can tell them apart)
- Let’s pause and think about what we know must be true
  1. Cardinal’s key > green’s key
AVL rotations

- Here is a Binary Search Tree whose keys I’m not going to show you
  - (but the nodes have colors/textures so you can tell them apart)
- Let’s pause and think about what we know must be true
  1. Cardinal’s key > green’s key
  2. Cardinal’s key > all 7 keys to its left!
AVL rotations

- Here is a Binary Search Tree whose keys I’m not going to show you
  - (but the nodes have colors/textures so you can tell them apart)
- Let’s pause and think about what we know must be true
  1. Cardinal’s key > green’s key
  2. Cardinal’s key > all 7 keys to its left!
  3. Green’s key < blue’s key < cardinal’s key
AVL rotations

- Here is a Binary Search Tree whose keys I’m not going to show you
  › (but the nodes have colors/textures so you can tell them apart)

- Let’s pause and think about what we know must be true
  1. Cardinal’s key > green’s key
  2. Cardinal’s key > all 7 keys to its left!
  3. Green’s key < blue’s key < cardinal’s key

- Those are just a few examples of the kind of reasoning you’ll want to use for this exercise…
AVL rotations

- **Your turn:** Which of the trees below are still in BST order? *(list all that apply)*

(A) ![Tree A]

(B) ![Tree B]

(C) ![Tree C]
AVL rotations

- 2/3 are actual AVL rotations!
- In this case, our BST started balanced, so the rotations made the less balanced. But also useful for balancing.
Left-Left AVL Rotation

- Right-Right is just the mirror image
Right-Left AVL Rotation

Original (valid but unbalanced BST):

- Left-Right is just the mirror image

Right-Left rotation (restores balance):
A few BST balance strategies

- AVL tree
  - Uses AVL rotations to guarantee balance

- Red-Black tree
  - Uses AVL rotations to guarantee balance is off by no more than a constant factor (longest path from root to leaf can be at most 2x the shortest path)

- Treap
  - Each node has *two* keys and a value, one is BST key, one is a min-heap key, both kinds of trees’ order properties are maintained (!!!)
  - Insert nodes according to BST keys and BST order
  - Then use AVL rotations to “bubble up” the newly inserted node as needed to restore the min-heap order property on the min-heap keys
  - What could be cooler than that, amirite? ♥ 😊 ♥ 😊
Red-Black trees

Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

(This is what guarantees “close” to balance)

Video: http://www.youtube.com/watch?v=vDHFF4wjWYU
Other fun types of BST

Splay tree
- Rather than only worrying about balance, Splay Tree dynamically readjusts based on **how often users search for an item**. Most commonly-searched items move towards the root, saving time
  - Example: if Google did this, “Bieber” would be near the root, and “splay tree” would be further down by the leaves

B-Tree
- Like BST, but a node can have many children, not just two
- More branching means an even “flatter” (smaller height) tree
- Used for huge databases
Tree Traversals!

These are for any binary trees, but we often do them on BSTs
What does this print? (assume we call traverse on the root node to start)

void traverse(Node* node) {
    if (node != NULL) {
        cout << node->key << " ";
        traverse(node->left);
        traverse(node->right);
    }
}

A. A B C D E F
B. A B D E C F
C. D B E F C A
D. D E B F C A
E. Other/none/more
What does this print?
(assume we call traverse on the root node to start)

```cpp
void traverse(Node* node) {
    if (node != NULL) {
        traverse(node->left);
        traverse(node->right);
        cout << node->key << " ";
    }
}
```

A. A B C D E F  
B. A B D E C F  
C. D B E F C A  
D. D E B F C A  
E. Other/none/more
What does this print?
(assume we call traverse on the root node to start)

```c
void traverse(Node* node) {
    if (node != NULL) {
        traverse(node->left);
        cout << node->key << " ";
        traverse(node->right);
    }
}
```

A. 1 2 4 5 8 9  
B. 1 4 2 9 8 5  
C. 5 2 1 4 8 9  
D. 5 2 8 1 4 9  
E. Other/none/more
Applications of Tree Traversals

Beautiful little things from an algorithms/theory standpoint, but they have a practical side too!
Traversals a very commonly-used tool in your CS toolkit

void traverse(Node* node) {
    if (node != NULL) {
        traverse(node->left);
        // "do something"
        traverse(node->right);
    }
}

- Customize and move the “do something,” and that’s the basis for dozens of algorithms and applications
Stanford Library Map

- Remember how when you iterate over the Stanford library Map you get the keys in sorted order?
  - (we used this for the word occurrence counting code example in class)
- Now you know why it can do that in $O(N)$ time!
  - **Stanford library Map is a BST**
  - **In-order traversal on BST!**
Applications of the traversals

- You are writing the **destructor** for a BST class. Given a pointer to the root, it needs to free each node. Which traversal would form the foundation of your destructor algorithm?

  A. Pre-order
  B. In-order
  C. Post-order
  D. Breadth-first
Applications of the traversals

- You are writing the destructor for a BST class. Given a pointer to the root, it needs to free each node. Which traversal would form the foundation of your destructor algorithm?

  A. Pre-order
  B. In-order
  C. Post-order
  D. Breadth-first

```c
void bstDestructorRecursiveHelper(Node *node) {
    if (node != NULL) {
        bstDestructorRecursiveHelper(node->left);
        bstDestructorRecursiveHelper(node->right);
        delete node; // post-order
    }
}
```
Breadth-First Tree Traversal

A somewhat different kind of traversal
How can we get code to print top-to-bottom, left-to-right order?

```cpp
void traverse(Node* node) {
    if (node != NULL) {
        cout << node->key << " ";
        traverse(node->left);
        traverse(node->right);
    }
}
```

You can’t do it by using this code and moving around the cout—we already tried moving the cout to all 3 possible places and it didn’t print in order

- You can but you use a **queue** instead of recursion
- “Breadth-first” search
- *Again we see this key theme of BFS (queue) vs DFS (stack/recursion)!*

Stanford University